

- Data that is stored directly in a variable
(not by a reference)
 - **Characters**
 - **Boolean values**
 - **Integers**
 - **Real values** (*as opposed to imaginary / complex numbers*)
- Not an instance of a class
- Operators on types executed directly by hardware

```
String s = new String("A String object");  
int i = 3;
```

Reference variable
stores reference to
object


s



Variables for
primitive data
types store value

i

3



"A String object"

	Reference variables	Primitive data variables
Type defined by	Class definition	Language
Value created by	<code>new</code> operator	Hardware
Value initialized by	Constructor	Hardware
Variable initialized by	Assignment of reference value	Assignment of primitive data

- Primitive data types do *not* have methods
- Instead have “*operators*”
- Examples for integers (int)
 - Addition: $i + 3$
 - Subtraction: $i - 5$
 - Multiplication: $3 * x$
 - Division: $22 / 7$ (<- not what you'd think!)
 - Remainder: $22 \% 7$

- Division – whole number division
 - 7 divides 22 three times
 - So $22 / 7 = 3$
 - Not 3.142857
- Remainder
 - Sometimes called “mod”
 - 7 divides 22 three times with remainder of 1
 - So $22 \% 7 = 1$
- Other operations yield what you would expect.

```
int w, x, y, z;
```

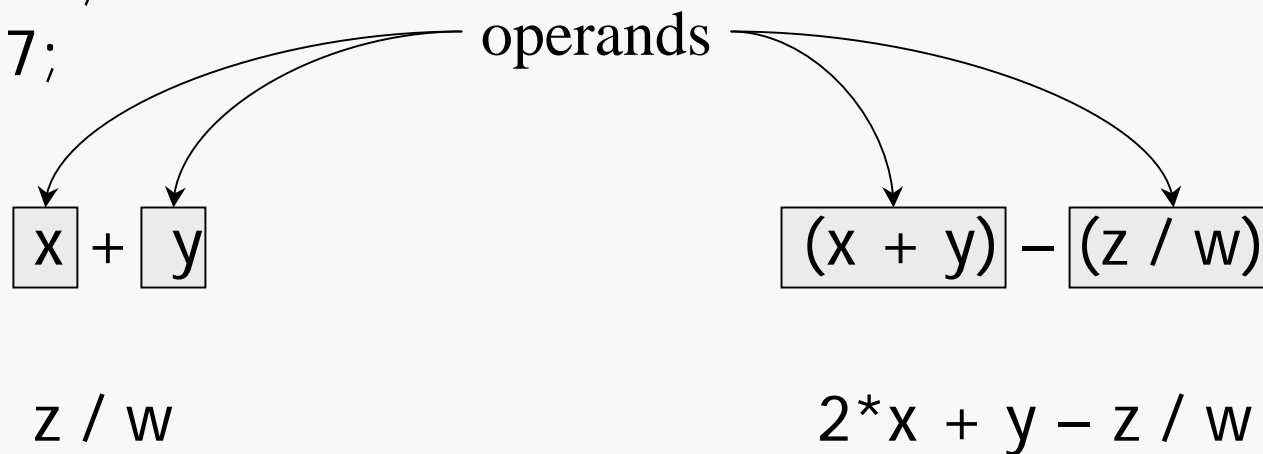
```
w = 4;
```

```
x = -3;
```

```
y = 12;
```

```
z = 7;
```

Expression: phrase that evaluates to a value.



- Determines order in which operators in expressions are evaluated
- For arithmetic operations, order is what you are used to from mathematics
 - Example: $3 + 5 * 2$ is 13 not 16
- Can override order with parenthesis
 - Example: $(3 + 5) * 2$ is 16
- As you learn new operators, you may want to use parenthesis when unsure of precedence

- Often one will write update statements like

```
earnedToDate = earnedToDate + monthsWages;  
salary = salary * 1.20;
```
- Java provides a shorthand, compound assignment for this

```
earnedToDate += monthsWages;  
salary *= 1.20;
```
- Use can help minimize errors in typing
- Assignment operators for the other standard integer operators are also provided by the language.

- Integer increment & decrement operators:

- Increment: add one to variable

```
x = x + 1;
```

```
x += 1;
```

```
x++;
```

- Decrement: subtract one from variable

```
x--;
```

Technically these are postfix increment and decrement operators because they follow the variable. There are also prefix versions of these operators that precede the variable (`++x;`). While there are execution differences, we will not go into them here. As long as the operators are used as single execution statements the execution results are the same.

(Briefly, we're not talking about integers...)

- Can use '+' to concatenate two strings
- Instead of writing
`s.concat(t)`
- Can write
`s + t`
- Useful when printing
`System.out.print("Name: " + beth.firstLast());`

The concatenation operator allows primitive types to be combined with string objects, automatically converting the result to a string.

- Different types allow different ranges of values
 - `byte` – 8 bit number: -128 to 127
 - `short` – 16 bit number: -32768 to 32767
 - `int` – 32 bit number \cong -2,000,000,000 to 2,000,000,000
 - `long` – 64 bit number \cong -8 quintillion to 8 quintillion
8,000,000,000,000,000,000
- Every bit doubles the range of numbers

- It is acceptable to assign a value from a smaller type to a variable of a larger type
 - byte to short
 - short to int
 - int to long

No data loss will occur.
Termed: “widening”

- But other way is not OK without a “**type cast**”

```
int x;  
long y = 6L;  
x = (int) y; //OK
```

Data loss may occur.
Termed: “shortening”

- Forced type conversion
- Syntax: **(type) expression**
- Examples:
 - If y is a **long** variable, **(int) y** is an **int** value
 - If b is a **short** variable, **(byte) b** is a **byte** value

Although allowed by the Java language data loss may occur due to “shortening”.

- Different types allow different ranges
 - `float` : 7 digits of precision
 - `double` : 15 digits of precision
- Caution: not all numbers in range can be represented by floating point numbers
 - `float` and `int` use same storage
 - But 1,234,567,089 is an int but not a float!

WHY???

- Integer values can be assigned to floating point variables – remember caution!
 - Some less significant digits may be lost due to limited precision.
- Floating point values can be assigned to integer variables with a cast
 - Truncation of the fractional, (decimal) quantity will occur.

- byte: number literal in valid range
- short: number literal in valid range
- int: number literal in valid range
- long: `32L` or `32l`
- float: `18f`, `18.f`, `18.0f`, `1.8e1f`, `.18E2f`
- double: `18`, `18.`, `18.0`, `1.8e1`, `.18E2`
also use code **d** or **D** for double, **f** or **F** for float

- To avoid “**magic numbers**” in code that reader might not understand
Example: literal constants in tax computations
- Literals will not change during execution.
- Use constant identifiers to declare these values

```
static final float SALES_TAX = 0.07f;
```

- Keyword **final** indicates that the value cannot change
- Use **static** so that all objects have access

- There are several classes that provide methods (static) that operate on primitive data
- Example: **class Math** provides general mathematical utilities
`int diffXY = Math.abs(x - y);`
- Other **wrapper classes** can hold values (for purposes needed later) and also provide functions
- Wrapper classes: **Byte**, **Short**, **Integer**, **Long**, **Float** and **Double**

- Use **Integer** method **parseInt** to convert String to int
 - **String s;**
 - **int numCalls;**
 - **s = kb.readLine(); //read line from input**
 - **numCalls = Integer.parseInt(s); //looks for int in s**
- Parsing means to separate a string into its parts
- Method requires an integer
 - String must be a number
 - If String contains anything but a valid integer an exception will be thrown.

■ Steps

- Read string
- Create **Double object** from string
- Extract **double value** from **Double object**

```
Double d = Double.valueOf(s);
```

```
double x = d.doubleValue();
```

■ Combine into one statement:

```
double x = Double.valueOf(s).doubleValue();
```

Double class serves as a wrapper class.
Double objects must be converted to primitive **double** values.

- Use integer types for counting
 - Number of times a step needs to be performed
 - Number of axles on a truck
 - Monetary values
- Use floating point types for measurement
 - Physical dimensions of objects
 - Graphical objects

- Rules for declaring variables for primitive data the same as for reference variables
- Can put primitive data variables in class declarations
- So, object may be made up of primitive data

```
class VitalStats {  
    public VitalStats (int a, int w, int hf, int hi,  
                        String hc, String ec, String r)  
        { ... }  
    //public methods go here  
    // ...  
  
    private int age, weight, heightFt, heightIn;  
    private String hairColor, eyeColor, race;  
}
```

■ Print spaces after strings to align next output under a heading label

- Define a String constant for spacing:

```
final String SPACES = "                ";
```

- Given the following sample heading labels:

```
Label1    Label2
```

- The first output field, (under **Label1**) has 10 columns for output.
- If the value printed only takes up 6 columns then 4 spaces must be printed to align the next output under the next heading label, **Label2**).
- The code below correctly pads after the value output for alignment:

```
myOutput.print(value1);  
myOutput.print(SPACES.substring(0, 10 - value1.length()));
```

- If value1 is numeric it must be converted to a String:

```
myOutput.print(value1);  
myOutput.print(SPACES.substring(0,10 - String.valueOf(value1).length()));
```

- The DecimalFormat class provides output format control for doubles.
 - Instantiate a DecimalFormat object passing a format string to the constructor:

```
DecimalFormat df = new DecimalFormat("#0.00");
```
 - The '#' symbol represents space for a digit, but the digit is not output if it is a leading or trailing zero.
 - The '0' symbol represents a digit. A zero is output even if it is a leading or trailing zero.
 - Other symbols, (e.g., '\$') may be included in the format string. Commas may be included to indicate digit grouping separations.
 - Example usage:

```
DecimalFormat df = new DecimalFormat("#0.00");  
Double d = 1.1;  
System.out.println(df.format(d));
```
 - Outputs:
_1.10
 - The underscore represents the leading space output by the '#' placeholder.